Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations*

Dragan Ivanović,[1] Manuel Carro,[1,2] and Manuel Hermenegildo [1,2]

[1] School of Computer Science, T. University of Madrid (UPM), Spain
(idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es)
[2] IMDEA Software Institute, Spain

Abstract. Service compositions put together loosely-coupled component services to perform more complex, higher level, or cross-organizational tasks in a platform-independent manner. Quality-of-Service (QoS) properties, such as execution time, availability, or cost, are critical for their usability, and permissible boundaries for their values are defined in Service Level Agreements (SLAs). We propose a method whereby constraints that model SLA conformance and violation are derived at any given point of the execution of a service composition. These constraints are generated using the structure of the composition and properties of the component services, which can be either known or empirically measured. Violation of these constraints means that the corresponding scenario is unfeasible, while satisfaction gives values for the constrained variables (start / end times for activities, or number of loop iterations) which make the scenario possible. These results can be used to perform optimized service matching or trigger preventive adaptation or healing.

Keywords: Service Orchestrations, Quality of Service, Service Level Agreements, Monitoring, Prediction, Constraints.

1 Introduction

Service-Oriented Computing is a paradigm that has been increasingly gaining ground as the basis for development of highly flexible, dynamic, and distributed service-based applications (SBAs). Key to the development of SBAs are service compositions, that allow the application designer to put together several loosely-coupled specialized component services, often provided and controlled by third parties, to perform more complex, higher-level, and/or cross-organizational tasks [7]. Trends in service-oriented application design indicate increased reliance on third-party services available on Internet [19].

In that context, quality of service (QoS) properties of individual services and their compositions are critical for overall usability. For externally offered services, service-level agreements (SLAs) define boundaries of permissible values for QoS attributes, such as execution time, availability, or cost. Potential and actual SLA violations can be avoided or mitigated using some form of adaptation (e.g., rebinding or changing

* The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube (Grant Agreement n° 215483). The authors were also partially supported by Spanish MEC project 2008-05624/TIN *DOVES* and CM project P2009/TIC/1465 (*PROMETIDOS*).

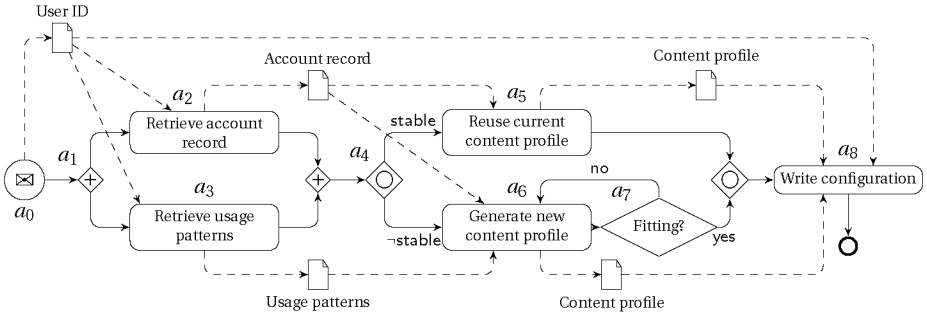

Fig. 1. An example orchestration to reconfigure content provided to a user.

the service selection preferences), or triggering structural changes both in the design and the running instance [7, 10]. For structurally constrained compositions of non-cyclic shape, flexible provisioning techniques have also been proposed [18].

Therefore, the task of analyzing and predicting QoS metrics for service compositions, both at design time and at the level of an executing instance, is of great theoretical and practical importance. Among the recently proposed approaches we can cite the application of statistical reasoning based on historical data (e.g., data mining) to predict likely SLA violations and their probable causes [15, 23], or to apply techniques related to model checking and online testing [10, 8].

In this paper, we take a different approach based on generating a constraint model for QoS metrics of an executing composition based on its structure, the semantics of its building blocks, and its current state of execution at a given moment. Previous works [4, 3, 13] also used the composition structure as the basis to derive properties thereof. In terms of results, instead of trying to find the most likely SLA conformance or violation scenario, we identify the possible cases of SLA conformance and violation at a given point of execution and infer conditions under which these may occur.

We consider service orchestrations, which are compositions with a centralized control flow. They may involve a wide range of workflow patterns [22] — including parallel flows, different splits/joins, loops, branches, etc. — and are usually expressed using some dedicated notation, such as BPMN [16], BPEL [14], Yawl [20] or DecSerFlow [21], or other adequate formalism. In this paper, we use abstract (but executable) notation for orchestrations from which we formulate a constraint satisfaction problem (CSP) [6, 1] that models the situation of SLA conformance or violation.

The rest of the paper proceeds as follows: Section 2 presents a motivating example. Section 3 then describes how the CSP can be automatically formulated on the basis of an orchestration continuation, to take into account the known assumptions about third-party components, as well as to include internal structural parameters of branches and loops. In Section 4 we present an experimental evaluation, Section 5 gives some implementation notes, and finally Section 6 presents conclusions.

2 Motivation

Consider a scenario where a provider of multimedia content (text, audio and video) needs to periodically update and reconfigure program streams offered to individual

clients (users), based on their historical usage patterns. That may require choosing between different mixtures of available streams (such as news, sport, entertainment, etc.) presented to a user, genres within them, and type of multimedia materials. The choice may depend on the frequency of use (casual vs. frequent users), user interests, and bandwidth adequate to serve different types of content (e.g. low quality vs. HD video). In such a scenario, the provider would run the reconfiguration process from time to time when serving user requests, although typically not for each access. Reconfiguration depends on other (usually back-end) administrative and analytic services, and should not cause noticeable glitches in content delivery. The SLA for the content delivery service does provide some window for running the reconfiguration process on top of it, but it is normally very restricted. Therefore, the running time of the reconfiguration process and its availability are of the utmost importance.

Fig. 1 depicts an example orchestration implementing the reconfiguration process, using BPMN notation [16]. It starts with the reception of user ID (activity a_0), which spawns in parallel (a_1) the retrieval of the users' account record (a_2) and the user's usage patterns (a_3). If the usage pattern is stable (a_4), the user's current content profile is reused (a_5). Otherwise, a new content profile is generated (a_6) based on the account record and the current usage patterns. For efficiency, first minor variations in content profile parameters are attempted; if these are not likely to fit the usage pattern (a_7), more radical changes are attempted, and so on. Finally, the content profile (either the current one or a new one) is written to the configuration database (a_8).

In this example, the configuration process may affect responsiveness of the main multimedia content delivery service, and therefore we want to continuously monitor and predict reconfiguration running time, having in mind the overall SLA. At any point in the execution of the reconfiguration orchestration, including its start, and within that particular context, there are a number of interesting objectives to aim at:

Predicting Certain SLA Violations: If we are able to predict that the orchestration cannot possibly meet the SLA constraints, then we can either abort it (effectively postponing the reconfiguration), or adapt it by switching to a simpler and/or more robust version. Conversely, if we are reasonably sure that the execution will be SLA-conformant, we can plan to use the potential slack in a productive way.

Predicting Possible SLA Violations: If we can predict that SLA violations may occur, but not necessarily so, and we can identify potential points of failure, then we can prepare, ahead of time, adequate adaptation and healing mechanisms, and/or try to decrease the risk of violation by using fail-safe component services.

Inferring the Necessary Preconditions: If we not only predict, but understand *why* an SLA violation may or must happen, we can use that information to identify bottlenecks, to develop criteria for selection of components, and to drive either runtime or design-time adaptation.

In this paper we present a unified constraint-based approach and analysis framework that makes it possible to perform runtime prediction of SLA violation / conformance for service orchestrations, based on monitoring information and a constraint model of an abstract semantics of the orchestration structure. Predictions are based

on and expressed in a form that describes the circumstances under which SLA violations and conformant executions of an orchestration may take place, which can be used to reason about the orchestration and its components.

3 Constraint-Based QoS Prediction

3.1 The General Prediction Framework

An SLA typically defines, among other things, which QoS attributes are relevant in the context of the provider-client contract, and what values of these QoS attributes are acceptable. For QoS attributes expressed as numbers on a measurement scale, QoS constraints given by an SLA are often expressed as ranges of permissible values for each attribute. More complex relationships between SLA attributes — such as trade-offs between cost and speed — can be devised, but in our analysis we will assume that the QoS constraints are given as lower and upper bounds on appropriate QoS metrics.

Furthermore, we will focus on an important subset of QoS metrics that are *monotonic* and *cumulative* in the sense that they express an amount of a physical or logical resource consumed by each activity in an orchestration, so that the amounts from subsequent activities add together into an aggregate metrics. Running time is an obvious example of a cumulative metrics, because consumed time is never recovered. In this paper we will assume, for simplicity, that metrics are accumulated by through addition (which is a fairly common case). Note that some metrics whose natural aggregation function is not addition can be easily mapped into additive metrics. For instance, the aggregation function for the availability (the probability of successful access) p of n subsequent operations can be calculated as $\prod_{i=1}^{n} p_i$, where $0 < p_i \leq 1$ is the availability of the i-th component. Using the transformation $\lambda = -\log p$, we can transform the original multiplicative metric of p into the additive $\lambda = \sum_i \lambda_i$.

An important feature of a cumulative QoS metrics is that, at any point in execution of an orchestration, its value can be calculated as a cumulative function (such as addition) of two components: the previously *accumulated metrics* and an estimate of the *pending metrics* for the remainder of the execution of the orchestration, until it finishes. For some metrics, their accumulated value needs to be be measured taking into account the history of the actual execution up to the current execution point (e.g. elapsed time from the start of execution), while for other metrics the current value at any execution point does not depend on the previous history. For example, in the case of availability the current metrics always represents "availability so far". Since it is being measured at some execution point which has obviously been reached, the probability p of being available up to the point of measure is 1 (and then $\lambda = 0$).

Let us present intuitively how accumulated metric values and a prediction for the rest of execution can be applied to predict SLA violations. We will use Fig. 2, taken from [12]. Points $\mathscr{A}$-$\mathscr{D}$ on the x-axis stand for the start, finish and two intermediate points in time during the execution of an orchestration. Let us assume that at the initial point $\mathscr{A}$ we have a prediction (solid line) for the QoS metrics for the rest of the execution. According to this prediction, the QoS at the finish falls under the limit *Max* given by some SLA. However, at point $\mathscr{B}$ we notice that some deviations

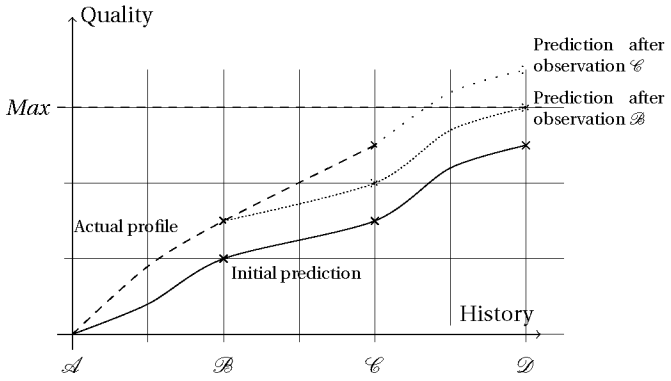

Fig. 2. Actual and predicted QoS throughout history.

have occurred up to that moment (the dashed line). Therefore, we adjust our prediction, which now seems to indicate borderline SLA compliance. At point $\mathscr{C}$, further measured deviations lead to another adjustment of the QoS prediction, this time indicating a likely violation of the SLA.

An important aspect of such prediction scheme is the existence of a time horizon between the detection of the possibility of an SLA violation and its actual occurrence. In our example, it is the period between $\mathscr{B}$ and the point of failure which lies somewhere between $\mathscr{C}$ and $\mathscr{D}$. This "window" makes it possible to warn about (potential) future SLA violations ahead of time. A prediction technique also needs to identify conditions that increase or decrease likelihood of an SLA violation, in order to filter false positives from true positives and thus increase the reliability of prediction. These conditions can be related to internal parameters of the orchestrations, such as the truth value of branching conditions or the number iterations in a loop. For our constraint-based approach, this will be illustrated in Section 3.5.

3.2 QoS Prediction Architecture

The architecture of the constraint-based QoS prediction framework is shown in Fig. 3. A process engine executes service orchestrations and interacts with external services by exchanging messages. In the process, it publishes lifecycle events such as signaling the start or end of a process, invocation of a component service, and reception of a reply. Also, from time to time, the process engine publishes the current point of execution of a running orchestration in the form of a continuation (explained in the following subsection). That is typically not done at each step, but at specific milestones such as service invocations, loop iterations and branches. Deciding how to determine the optimal granularity for publishing points is a matter for future work.

The events published by the process engine are sent via an event bus. The constraint-based QoS predictor can be connected to that bus and listen to lifecycle events (or a subset of events of interest). When a continuation is published, it is pushed by the event bus to the predictor. The predictor performs the analysis, and publishes QoS predictions back to the event bus, together with QoS metric bounds inferred by the analysis. That information can be accessed by an adaptation mechanism, which

can use the published predictions and the QoS metrics to prepare adequate adaptation actions on the orchestration definition, an executing instance, or both. Such adaptation actions may include, among other things, selection of components to minimize the risk of failure, changes in the structure of the process, or intervention on the orchestration data.

3.3 Representing Orchestrations and their Continuations

In order to estimate how much the remainder of the execution can contribute to a given QoS metrics, we need to have some knowledge about where in the execution we are placed — or, more precisely, what remains to be executed: it is the orchestration activities yet to be executed which need to be taken into account to predict the remainder of the metric value. In our case we represent this still-not-executed part of the orchestration explicitly, in the form of a *continuation*. A continuation [17] is an abstract object (such as a set of data structures or a function) that represents the control state of a computation — i.e., the precise execution point of a program (including the associated data) and whatever remains to be executed.

In our case we are interested in continuations of running instances of orchestrations. A continuation is always implicit in the state of a process engine, even when the chosen programming language does not make it accessible as such: it is determined, for example, by the activity being executed, the representation of the orchestration and the data in the orchestraton. In our approach, we rely on keeping available at all moments an explicit representation of the continuation, inspect its structure (which in general becomes progressively simpler as execution proceeds) and use it to generate constraints which model the conditions under which the execution can meet / not meet the QoS stated in the SLA.

The (simplified) abstract syntax we will use is shown in Fig. 4. It is based on the concrete syntax used by a prototype orchestration engine which we developed as ex-

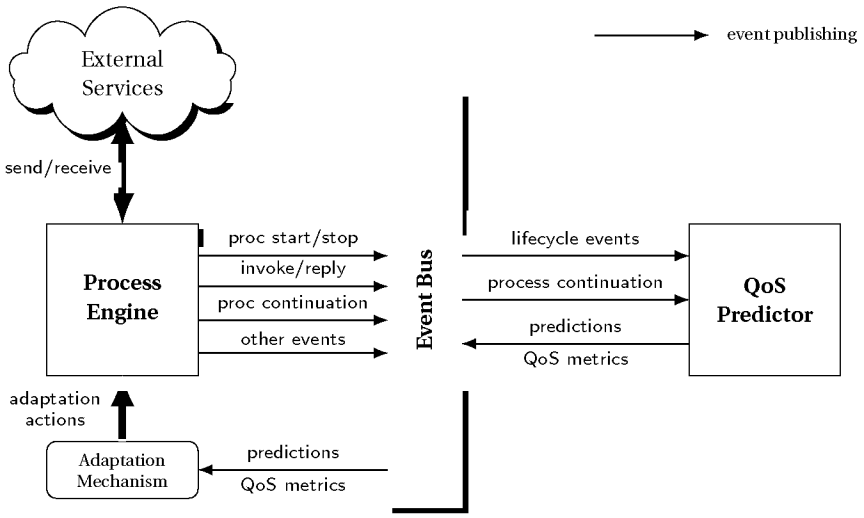

Fig. 3. Architecture of the QoS Analysis Framework.

$$
\begin{array}{lll}
\textit{continuation} ::= a\,. & & \\
\quad a, a_1, a_2 ::= & \{\textit{elementary operation}\} & \text{(elementary operation)} \\
& \mid a_1\,,\, a_2 & \text{(sequence)} \\
& \mid (\{\textit{cond}\} \rightarrow a_1\,;\, a_2) & \text{(if-then-else)} \\
& \mid a_1 \wedge a_2 & \text{(and-join)} \\
& \mid a_1 \vee a_2 & \text{(or-join)} \\
& \mid \texttt{while}(\{\textit{cond}\},\, a) & \text{(while loop)} \\
& \mid \texttt{foreach}(x,\, \textit{list},\, a) & \text{(list comprehension)} \\
& \mid \texttt{invoke}(\textit{partner},\, \textit{out},\, \textit{in}) & \text{(invoke a service)} \\
& \mid \texttt{reply}(\textit{out}) & \text{(send a reply)} \\
& \mid \texttt{relax} & \text{(do nothing)} \\
& \mid \texttt{stop} & \text{(finish)}
\end{array}
$$

Fig. 4. Abstract syntax for orchestrations.

perimentation base for this paper and that uses Prolog as the language to express branch and loop conditions and elementary operations. A *simple activity* represents a basic unit of work, such as a calculation or assignment. Similarly, *cond* encodes a logical condition that is used for if-then-else branching or `while` loop iteration. List comprehension is simplified using `foreach`. Communication with the environment is done using `invoke` and `reply`. Besides sequences, both parallel OR and AND splits/joins are supported. Most BPMN constructs can be translated straightforwardly. A translation of the example process from Fig. 1 (with some low-level details omitted) is shown on the left of Fig. 5.

The continuation at every point of the execution of Fig. 5 is not explicit in the orchestration representation, but is rather kept by the interpreter which executes it (which we do not have space to describe in detail in this paper). This continuation represents what is left to execute after every computation step, and is updated every time a step is taken. For instance, after taking the *else* branch in the orchestration from Fig. 5 (left), the continuation is a sequence of activities in lines 6-9, 11 and 12.

3.4 Deriving QoS Constraints from Continuations

A *constraint* is a relation that restricts values of variables that, in our case, represent values of QoS metrics associated with the constructs in the orchestration and their

```
1  (  invoke(account_svc, UserID, AccRec)                    500 ≤ T1 ≤ 800          (assumption: account_svc)
2  ∧  invoke(usage_svc, UserID, UsagePatt)                   200 ≤ T2 ≤ 500          (assumption: usage_svc)
3  ),                                                        T3 = max(T1, T2)        (∧-join)
4  (  stable(UsagePatt)}                                     Cond ∈ {0, 1}, 0 ≤ T4 ≤ 10   (condition)
5  →  invoke(reuse_svc, AccRec, Profile)                     100 ≤ T5 ≤ 400          (assumption: reuse_svc)
6  ;  invoke(gen_svc, (AccRec, UsagePatt), Profile),         200 ≤ T6 ≤ 600          (assumption: gen_svc)
7     while( {unfit(Profile)},                               k ∈ N, 0 ≤ T7 ≤ 10      (while condition)
8       invoke(gen_svc, (AccRec, UsagePatt), Profile) )      200 ≤ T8 ≤ 600          (assumption: gen_svc)
9     )                                                     T9 = k × (T7 + T8) + T7   (while duration)
10 ),                                  (Cond = 1 ∧ T10 = T4 + T5) ∨ (Cond = 0 ∧ T10 = T4 + T6 + T9)   (if)
11 invoke(conf_svc, (UserID, Profile), _),                   100 ≤ T11 ≤ 300         (assumption: conf_svc)
12 stop.                                                     T = T3 + T10 + T11      (total running time)
```

Fig. 5. Orchestration for Fig. 1 (left) and its associated running time constraints (right).

basic components. The particular relations which are generated depend both on the QoS metric that is to be captured and on the structure of the continuation. In our approach, after deriving the constraints from the structure of the given continuation, constraint solving techniques (see Section 3.6) are used to infer admissible ranges for variables that lead to either SLA satisfaction or violation.

We require that these constraints lead to a conservative prediction of QoS fulfillment: under the assumption that our knowledge about the QoS characteristics of the basic orchestration components (i.e., atomic activities or external services) is correct,[3] we want that any prediction we make about the conformance of an execution w.r.t. the stated SLA is also correct. In this direction, we make no assumptions on the (in)dependence of behavior of individual components. I.e., if the behavior of two external services seems to be strongly linked (because of e.g. past history), we do not take this apparent correlation into account for the sake of prediction safety. Such information, if available, could be added to try to make predictions more precise: for example, given that some service took less time than expected to answer, we might assume that the same is going to happen to some other service which is apparently historically related. While this seems to help in making predictions more accurate, it also makes them potentially unsafe.

We illustrate constraint derivation with two metrics: *running time* and *availability*. For a continuation consisting of a (complex) activity a representing the remainder of the execution, the total running time of the orchestration is a sum of the elapsed time since the start T_a and the pending time $T(a)$. The total availability is equal to the pending availability $\lambda(a)$, as explained before. We derive $T(a)$ and $\lambda(a)$ structurally, and then constrain them against the SLA limit: $T_{\max}$ for the maximal allowed execution time by and $\lambda_{\max}$ for the negative logarithm of the minimal allowed availability (see Section 3.1). The resulting constraints:

For SLA conformance: $T_a + T(a) \leq T_{\max}$ and $\lambda(a) \leq \lambda_{\max}$.
For SLA violation: $T_a + T(a) > T_{\max}$ and $\lambda(a) > \lambda_{\max}$.

are solved to obtain the (approximate, but safe) ranges for $T(a)$ and $\lambda(a)$, and thus for the total QoS, for the two cases of conformance and violation, respectively.

We generate the above constraints by formulating a constraint for each simple activity contained inside a (usually relating the value of the QoS metric for the activity with its expected bounds) and combining these constraints (using disjunctions and conjunctions according to the structure of a) into a larger constraint which provides bounds for $T(a)$. The right hand side of Fig. 5 shows the set of constraints corresponding to the process on the left. We will now detail how constraints for simple and complex activities are generated.

Simple activities. For a simple activity a — a simple operation, `relax` or `stop` — and simple operations (in curly braces), the assumption is that they include only elementary constructs and do not entail complex computations. A lower bound for

[3] Note that in reality this knowledge is always inexact and subject to dynamic changes. However, we are putting ourselves in the situation that this knowledge is exact, and we want to ensure that, at least in this optimistic situation, the constraints we generate meet safeness requirements.

this is always $T(a) \geq 0$, and an upper bound depends on the execution environment (computer clock, CPU, etc.). It is usually on the order of microseconds, and should be experimentally determined for each architecture. In the example we have put some reasonable limits, which do not necessarily reflect a real situation. As for the availability, since no external components are involved, in this case we have $\lambda(a) = 0$.

Sequences. Since we are considering cumulative metrics,[4] the metric values are accumulated for the case of sequences: for sequence $a \equiv a_1, a_2$ we have $T(a) = T(a_1) + T(a_2)$ and $\lambda(a) = \lambda(a_1) + \lambda(a_2)$.

Service invocations. For an activity a that is an `invoke` to an external service, for both the running time $T(a)$ and the availability $\lambda(a)$ the analyzer needs to rely on empirically or analytically derived estimates, which include the local message handling and network delivery. In our approach, we deal with the ranges of *possible* values, rather than with *probable* or *expected* values. That means that in absence of any information, we simply have $T(a) \geq 0$ and $\lambda(a) \geq 0$, but the upper bounds on $T(a)$ and $\lambda(a)$, if known, must be safe, or else the prediction will be too optimistic and fail to detect some cases of possible SLA violations.

Parallel flows. In the case of a parallel flow $a \equiv a_1 \wedge a_2$, $T(a)$ must lay somewhere between $\max(T(a_1), T(a_2))$, when a_1 and a_2 run fully in parallel, and $T(a_1) + T(a_2)$, which is the worst, sequential case of execution. Therefore, it is safe to take

$$\max(T(a_1), T(a_2)) \leq T(a) \leq T(a_1) + T(a_2)$$

as a conservative approximation.

This approximation can however be too cautions and may lead to overly pessimistic estimates. If we have additional information about the semantics of the orchestration language and the implementation of the execution engine, we can refine the estimate for $T(a)$. For instance, if the execution of local activities is single threaded, while external services invocations are ensured to run in parallel, we can use the following scheme. Consider the case where a_1 and a_2 are sequences ending with an `invoke` activity, i.e., $a_1 \equiv a_{11}, a_{12}, \ldots, a_{1k}, a_1^*$ and we call $a_1' \equiv a_{11}, a_{12}, \ldots, a_{1k}$ (respectively for a_2). We will assume that a_1' and a_2' are sequences of activities to be executed locally by a single thread, even if they appear in different branches of the flow, while a_1^* and a_2^* can be executed remotely in parallel. In this case, the total estimated time for the flow is

$$\max\big(T(a_1') + T(a_1^*), T(a_2') + T(a_2^*)\big) \leq T(a) \leq T(a_1') + T(a_2') + \max\big(T(a_1^*), T(a_2^*)\big)$$

If, say, a_1^* is not an external `invoke`, but a_2^* is, then $T(a_1^*)$ is part of $T(a_1')$. If neither a_1^* nor a_2^* are external `invokes`, then simply $T(a_1^*) = T(a_2^*) = 0$. This structural analysis can of course be easily extended to more than two parallel flows. The running time of an OR-parallel flow can be conservatively approximated using the case of AND-parallelism.

From the point of view of availability, parallel flows do not affect the total risk of failure, since the total availability depends on availability of all used components, regardless of their order of execution. Therefore, for $a \equiv a_1 \wedge a_2$ or $a \equiv a_1 \vee a_2$, we have $\lambda(a) = \lambda(a_1) + \lambda(a_2)$.

[4] Or those that can be converted into a cumulative (e.g. additive) equivalents.

Conditionals. For a conditional $a \equiv (\{cond\} \rightarrow a_1 \ ; \ a_2)$, where a_1 is the *then* part and a_2 is the *else* part, the metric depends on how the condition is evaluated. We introduce a Boolean variable b_{cond} to represent the result of the condition evaluation, so that we can state the following disjunctive constraint: either (1) $b_{cond} = 1$ and $T(a) = T(\{cond\}) + T(a_1)$, $\lambda(a) = \lambda(a_1)$, or (2) $b_{cond} = 0$ and $T(a) = T(\{cond\}) + T(a_2)$, $\lambda(a) = \lambda(a_2)$. The value of b_{cond} is generally unknown, but can be constrained to either 0 or 1 as the result of constraint solving. This makes it explicit that either the then or the else part can be taken, but not both.

Loops. In case of a loop a — `while` or `foreach` with body a_1 — we introduce an integer variable $k_a \geq 0$ that stands for the number of loop iterations. Then, we have $T(a) = k_a \times (T(\{cond\}) + T(a_1)) + T(\{cond\})$ and $\lambda(a) = k_a \times \lambda(a_1)$. The actual value of k_a is generally unknown, but its inclusion into the constraints allows us to reason about the maximal or minimal number of loop iterations that lead to SLA compliance or violation.

3.5 Using Computational Cost Functions

To improve the precision of the predictions, the constraint-based predictor is able to use computational cost functions for service orchestrations [13], which, in this case, express lower and upper bounds of the number of loop iterations as a function of the input data to the orchestration. These computation cost functions may be automatically inferred at the start of an orchestration, statically determined at design time, or manually asserted for known cases. The inference of the computation cost functions depends on the semantics of the workflow constructs and the (sub-)language of conditions and elementary operations in which the orchestration is expressed.

If computation cost functions are available, the default constraint for the number of iterations of loop a ($0 \leq k_a$) can be strengthened to $\ell \leq k_a \leq u \wedge 0 \leq k_a$, where ℓ and u are, respectively, lower and upper bounds on the number of iterations, which depend on the actual values of the input data. In the absence of one (or both) of the bounds, the corresponding constraint is simply not generated (as in Fig. 5, right).

3.6 Solving the Constraints

The constraints derived from the orchestration continuation relate the QoS metrics for the entire continuation with those of individual activities, component services, Boolean results of evaluating the conditions, the number of loop iterations, and the limits from the SLA. As such, they represent a constraint satisfaction problem [6] that can be solved for values of the constrained variables, which, in our case, include QoS metrics, Boolean conditions and loop iteration counters. Depending on the type of problem and the particular constraint solver used, solving the CSP may involve several iterations of *constraint propagation* and *problem splitting* [6, 1], which are used to reduce the equations in the original CSP to a series of simpler ones, before attempting to assign to the constrained variables values that satisfy the constraints.

In our case, we use the *interval constraints* (*ic*) solver from the *ECL^{i}PS^e* Constraint Logic Programming (CLP) system [2, 5]. The underlying Prolog subsystem of

ECL^iPS^e is used for constructing the constraints from a continuation, handling information on QoS metrics of component services, and reporting the results. The solver handles constrained variables over bound and unbound integer (discrete) and real number (dense) domains. The values of the constrained variables are represented as (possibly unbound) real or integer intervals. Integer variables with bounded domains are handled in a manner similar to finite domain solvers [6]. The solver directly supports disjunctive constraints (which we use for conditionals) and reified (Boolean valued) constraints.

The solver produces results given as bounds on values of the constrained variables, obtained from propagation of arithmetic constraints, or fails if the constraints cannot be satisfied. In our case, as mentioned before, we always solve two CSPs, one modeling SLA conformance and another one modeling SLA violation.

The constraint solver is complete, i.e., it does not discard feasible solutions. Therefore, upon constraint satisfaction, the answer intervals for the variables include all admissible values, and values outside these intervals cannot possibly satisfy the constraints. On the other hand, it may be that some combinations of values inside the answer intervals do not satisfy the constraints. Let us see an example: the constraint $0 \leq T(a_1) + T(a_2) \leq 100$ has as answer $T(a_1) \in [0..100] \wedge T(a_2) \in [0..100]$. This contains all feasible solutions (for example, $T(a_1) = 0 \wedge T(a_2) = 100$) but also combinations of values which do not satisfy the constraints (for example, $T(a_1) = 50 \wedge T(a_2) = 51$). Of course, if the latter values are fed into the constraint solver together with the initial constraint, the constraint solver will determine that the system is unsolvable.

4 Experimental Evaluations

Table 1 shows the results of running our QoS prediction framework applied to the orchestration in Fig. 5 (corresponding to the workflow in Fig. 1) and using execution time as QoS metric. The assumptions on ranges for the invocations of external services are shown at the bottom. These ranges would be updated by the QoS predictor based on the observation of invoke/reply events published by the process engine. Note that we are only concerned with the range of *possible* running times for each component, not the probability distributions within these ranges, and therefore we only need only to adjust the boundaries of the corresponding ranges.

The top part of Table 1 shows the results for the case of an unbound number of `while` loop iterations, which is the default if no additional information is provided. A series of successive assumed running time limits (500, 750, 1 500 and 3 000 ms) was considered, and both the SLA compliance (*success*) and *violation* results are shown. The meaning of the rest of the rows are as follows:

duration shows the predicted running time ranges for the orchestration in ms.

cond(if) is a Boolean value showing the possible evaluations of the condition in the conditional (1 for the "then" branch and 0 for the "else" branch).

iter(while) shows the range of possible iteration counts in the `while` loop (corresponding to the repetition after testing the condition in the "else" branch).

E.C.D.T. *earliest certain detection times*: the earliest time at which a certain violation or success can be detected.

Case 1: Unconstrained iterations									
		Successive running time SLA ranges							
		0 ms .. 500 ms		500 ms .. 750 ms		750 ms .. 1 500 ms		1 500 ms .. 3 000 ms	
Variable	Metrics	*success*	*violation*	*success*	*violation*	*success*	*violation*	*success*	*violation*
duration	ms	—	600 .. +∞	600 .. 750	750 .. +∞	750 .. 1 500	1 500 .. +∞	1 500 .. 3 000	3 000 .. +∞
cond(if)	bool	—	0 .. 1	1	0 .. 1	0 .. 1	0	0	0
iter(while)	nat	—	0 .. +∞	—	0 .. +∞	0 .. +∞	0 .. +∞	0 .. 11	3 .. +∞
E.C.D.T.	ms	—	0	500	450	500	1 200	700	2 700
% E.C.D.T.		—	0%	66%	60%	33%	80%	23%	90%
Lead	ms	—	500	250	300	1 000	300	2 300	300

Case 2: Between 1 and 10 iterations									
		Successive running time SLA ranges							
		0 ms .. 500 ms		500 ms .. 750 ms		750 ms .. 1 500 ms		1 500 ms .. 3 000 ms	
Variable	Metrics	*success*	*violation*	*success*	*violation*	*success*	*violation*	*success*	*violation*
duration	ms	—	600 .. 7 820	600 .. 750	750 .. 7 820	750 .. 1 500	1 500 .. 7 820	1 500 .. 3 000	3 000 .. 7 820
cond(if)	bool	—	0 .. 1	1	0 .. 1	0 .. 1	0	0	0
iter(while)	nat	—	1 .. 10	—	1 .. 10	1 .. 10	1 .. 10	1 .. 10	3 .. 10
E.C.D.T.	ms	—	0	500	250	500	1 000	900	2 500
% E.C.D.T.		—	0%	66%	33%	33%	66%	30%	83%
Lead	ms	—	500	250	500	1 000	500	2 100	500

	Component running time assumptions					
	local op.	account_svc	usage_svc	reuse_svc	gen_svc	conf_svc
Running time (ms)	0 ms .. 10 ms	500 ms .. 800 ms	200 ms .. 500 ms	100 ms .. 400 ms	200 ms .. 600 ms	100 ms .. 300 ms

Table 1. Sample QoS prediction results.

% E.C.D.T. percentage of the total (maximum) execution time which elapsed up to the E.C.D.T.

lead time between E.C.D.T. and the closest moment in which the orchestration can finish (i.e., the shortest time span to react in the worst case).

The results show that the lowest limit of 500 ms could not be met under the initial assumptions regarding execution times for atomic activities and external services. The 750 ms limit can be met, if the conditional evaluates to 1, meaning that the `while` loop is avoided. The 1 500 ms limit can be met in both cases of the conditional, but can be violated only for the case of taking the "else" branch. Finally, for the range of running times between 1 500 ms and 3 000 ms, the prediction shows that, under the given assumptions, the only possible situation for both compliance and violation is taking the "else" branch, with the number of iterations in the range 0 .. 11 and 3 .. +∞, respectively. Note that for the latter limit, between 0 and 2 iterations guarantees compliance, and more than 11 iterations guarantees violation of the limit. An adaptation mechanism can, use these predictions to prepare and trigger adaptation actions that may prevent, minimize, or compensate for possible SLA violations ahead of time.

The earliest time at which a success or violation can be predicted depend on the particular execution. Let us look at an example: in Table 1, Case 1, columns "750 ms .. 1500 ms", successes have been detected at 500 ms and SLA violations at 1200 ms. The reason that successes have been detected before violations is that these correspond to different executions: in the case of violation, the "else" branch (with the loop) has been taken, it is detected that there will be a violation after some iterations. On the other hand, if the "then" branch is taken, certainty of success is immediately detected, as there are no loops to be taken. With this interpretation in mind,

the constraint-based predictor is able to detect SLA violation with certainty up to between 300 and 500 ms in advance, while SLA conformance can be detected as early as after 500 or 700 ms of running time. In relative terms, SLA conformance has been detected in the experiments when only between a 23% and a 66% of the maximum execution time has elapsed, and SLA violations have been detected in some cases when only a 60% of the execution has elapsed.

The middle part of Table 1 shows a hypothetical case where, based on input data and computational cost functions, the predictor is able to infer that the actual number of loop iterations, in case the "else" branch is taken, must fall between 1 and 10. The results follow the same pattern as in the first case, but this time the predictor is able to infer that the duration of the orchestration under the assumptions must fall between 600 and 7 820 ms. This inferred running time range for the orchestration can be used by other parts of the runtime system (including predictors themselves) to update their QoS metrics assumptions on the deployed components. Note that the guarantee of at least one loop iteration increases the lead for the earliest certain detection of violations to 500 ms.

The average net time for performing one running time limit compliance/ violation prediction depicted in Table 1 (not counting the time for sending and receiving data over the network), based on the average from 10 000 executions, was 0.574 ms on a small end-user non-dedicated machine.[5]

5 Implementation Notes

We have tested the approach using a prototype implementation of the architecture from Fig. 3, which includes the process engine, the QoS predictors, and the event bus, organized as a distributed and scalable system of components that communicate using reliable messaging. The tests included deployments on Linux and Mac OS X 32 and 64 bit platforms.

In our running prototype, the QoS predictors are implemented in ECL^iPS^e Constraint Logic Programming system, while the process engine (that executes orchestrations) is implemented in Ciao Prolog [9]. Both Prolog dialects support a variety of constraint logic programming techniques, but have, at the moment, slightly different orientation and strong points. ECL^iPS^e provides very robust, industrial-scale constraint solvers which can easily handle very complex problems involving thousands of constraints and variables, while Ciao is a flexible multi-paradigm programming environment with sophisticated support for concurrency. Fortunately the fact that they are both Prolog-based systems greatly facilitates interfacing and putting together the required architecture.

In our prototype, the language in Fig. 4 is used to define service orchestrations and to maintain instance control state throughout execution, so that there is no additional overhead in communicating continuations to QoS predictors, other than message transfer times. Also, any adaptation that changes the orchestration structure for

[5] The tests were run on a 32-bit 2GHz Intel Core Duo notebook with 2GB of RAM, running Mac OS X 10.6.7 and ECL^iPS^e version 6.0_167.

a running instance can be simply implemented by replacing one continuation with another.

The messaging subsystem is implemented using ZeroMQ [11], which provides fast and reliable multi-part binary message exchange primitives on top of TCP networking and IPC subsystems, including request-reply, push-pull and publish-subscribe patterns. We have developed Prolog (Ciao and ECL^iPS^e) bindings to ZeroMQ with data (term) serialization that provide transparent higher-level data exchange primitives.

6 Conclusions

We have devised and implemented a method which makes it possible to predict possible situations of SLA conformance and violation, and to obtain information on the internal parameters of the orchestration (branch conditions, loop iterations) that may occur in these situations. The method is based on modeling QoS metrics of a service orchestration using constraints, based on assumptions on the behavior of the orchestration components. That analysis can, in principle, be applied at each step in an orchestration based on the current continuation. This allows periodic or continuous updating of the predicted bounds for QoS metrics for the orchestration and therefore a continuous assessing of conformance to SLA, which can be useful for proactive adaptation and self-healing. This approach can be combined with automatically inferred computational cost functions for service orchestrations, which can express the bounds of internal parameters (such as loop iterations) as functions of input data given to the orchestration instance, to provide a higher level of prediction precision. We have implemented the method in a prototype and reported some efficiency results.

Our future work will concentrate on making the implementation of all elements of the QoS prediction architecture laid out in this paper more complete and robust, including the process engine, beyond the prototype stage. We also plan to add support for different execution engines, targeting specifically those that have well-defined interfaces for event-listening plugins or can be adapted accordingly (e.g. because the implementation is open-source).

References

1. K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. Krzysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming Using ECLIPSE.* Cambridge University Press, 2007.
3. J. Cardoso. About the Data-Flow Complexity of Web Processes. In *6th International Workshop on Business Process Modeling, Development, and Support: Business Processes and Support Systems: Design for Flexibility*, pages 67–74, 2005.
4. Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281 – 308, 2004.
5. Cisco Systems. *ECLIPSE User Manual*, 2006.
6. Rina Dechter. *Constraint Processing.* Morgan Kauffman Publishers, 2003.

7. Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15:313–341, 2008. 10.1007/s10515-008-0032-x.
8. Dimitris Dranidis, Andreas Metzger, and Dimitrios Kourtesis. Enabling proactive adaptation through just-in-time testing of conversational services. In Elisabetta Di Nitto and Ramin Yahyapour, editors, *ServiceWave*, volume 6481 of *Lecture Notes in Computer Science*, pages 63–75. Springer, 2010.
9. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 2012. http://arxiv.org/abs/1102.5497.
10. Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In Petri Mähönen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science*, pages 122–133. Springer Berlin / Heidelberg, 2008.
11. iMatix Corporation. *0MQ - The Reference Manual, version 2.1*, June 2011.
12. D. Ivanović, M. Carro, and M. Hermenegildo. An Initial Proposal for Data-Aware Resource Analysis of Orchestrations with Applications to Predictive Monitoring. In Asit Dan, Frédéric Gittler, and Farouk Toumani, editors, *International Workshops, ICSOC/ServiceWave 2009, Revised Selected Papers*, number 6275 in LNCS. Springer, September 2010.
13. D. Ivanović, M. Carro, and M. Hermenegildo. Towards Data-Aware QoS-Driven Adaptation for Service Orchestrations. In *Proceedings of the 2010 IEEE International Conference on Web Services (ICWS 2010), Miami, FL, USA, 5-10 July 2010*. IEEE, 2010.
14. D. Jordan and et. al. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, et. al, 2007.
15. Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime prediction of service level agreement violations for composite services. In Asit Dan, Frederic Gittler, and Farouk Toumani, editors, *ICSOC/ServiceWave Workshops*, volume 6275 of *Lecture Notes in Computer Science*, pages 176–186, 2009.
16. Object Management Group. *Business Process Modeling Notation (BPMN), Version 1.2*, January 2009.
17. John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation Journal*, 6:233–247, 1993.
18. Sebastian Stein, Terry R. Payne, and Nicholas R. Jennings. Robust execution of service workflows using redundancy and advance reservations. *IEEE T. Services Computing*, 4(2):125–139, 2011.
19. G. Tselentis, J. Dominigue, A. Galis, A. Gavras, and D. Hausheer. *Towards the Future Internet: A European Research Perspective*. IOS Press, Amsterdam, The Netherlands, 2009.
20. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, June 2005.
21. Wil van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, 2006.
22. Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
23. Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar, and Frank Leymann. Monitoring and analyzing influential factors of business process performance. In *EDOC*, pages 141–150. IEEE Computer Society, 2009.